An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads

Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki,
Akio Nishimura, Yoshimori Nakase and Teiji Nishizawa

Media Research Laboratory,
Matsushita Electric Industrial Co., Ltd.
1006 Kadoma Osaka. 571 Japan
(hirata@isl.mei.co.jp)

Abstract

In this paper, we propose a multithreaded processor architecture which improves machine throughput. In our processor architecture, instructions from different threads (not a single thread) are issued simultaneously to multiple functional units, and these instructions can begin execution unless there are functional unit conflicts. This parallel execution scheme greatly improves the utilization of the functional unit. Simulation results show that by executing two and four threads in parallel on a nine-functional-unit processor, a 2.02 and a 3.72 times speed-up, respectively, can be achieved over a conventional RISC processor.

Our architecture is also applicable to the efficient execution of a single loop. In order to control functional unit conflicts between loop iterations, we have developed a new static code scheduling technique. Another loop execution scheme, by using the multiple control flow mechanism of our architecture, makes it possible to parallelize loops which are difficult to parallelize in vector or VLIW machines.

1 Introduction

We are currently developing an integrated visualization system which creates virtual worlds and represents them through computer graphics. However, the generation of high quality images requires great processing power. Furthermore, in order to model the real world as faithfully as possible, intensive numerical computations are also needed. In this paper, we present a processor architecture[1] used as the base processor in a parallel machine which could run such a graphics system.

In the field of computer graphics, ray-tracing and radiosity are very famous algorithms for generating realistic images. In these algorithms, intersection tests account for a large part of the processing time for the whole program. This test has inherent coarse-grained parallelism and can easily create many threads(parallel processes) to be executed on multiprocessing systems. Each thread, however, executes a number of data accesses and conditional branches. In the case of a distributed shared memory system, low processor utilization can result from long latencies due to remote memory access. On the other hand, low utilization of functional units within a processor can result from inter-instruction dependencies and functional operation delays. Another obstacle to the optimization of the single thread execution arises if the past performance of a repeatedly executed conditional branch does not help in predicting the target of future executions.

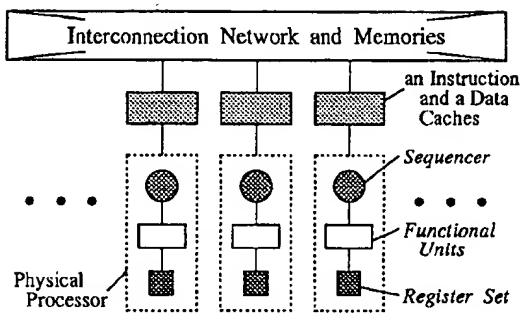
In order to overcome these problems, we introduced two kinds of multithreading techniques into our processor architecture: *concurrent multithreading* and *parallel multithreading*. The concurrent multithreading technique attempts to remain active during long latencies due to remote memory access. When a thread encounters an absence of data, the processor rapidly switches between threads. On the other hand, parallel multithreading within a processor is a latency-hiding technique at the instruction level. When an instruction from a thread is not able to be issued because of either a control or data dependency within the thread, an independent instruction from another thread is executed. This technique is especially effective in a deeply pipelined processor.

Our goal is to achieve an efficient and cost-effective processor design which is oriented specifically for use as an elementary processor in a large scale multiprocessor system rather than for use in an uniprocessor system. The idea of parallel multithreading arose from hardware resource optimization where several processors are

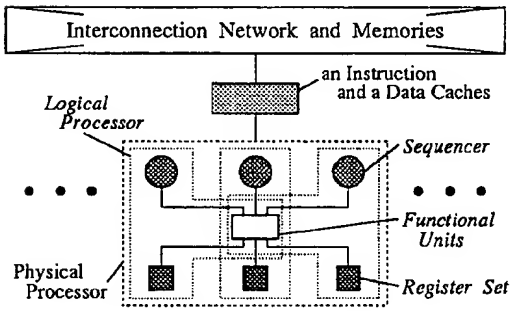
united and reorganized so that functional units could be fully used. Figure 1(a) shows a general multiprocessor system organization. Multiple threads are executed on each physical processor. For example, assume that the utilization of the busiest functional unit of a processor in Figure 1(a) is about 30% because of the instruction level dependency. The utilization of the functional unit is defined as $U = \frac{N \cdot L}{T} \times 100[\%]$, where N is the number of invocations of the unit, L is an issue latency of the unit, and T represents the total execution cycles of the program. In this case, three processors could be united into one as shown in Figure 1(b), so that the utilization of the busiest functional unit could be expected to be improved nearly to 30×3=90%. Consequently, on the united processor, multiple instructions from different threads are issued simultaneously and executed unless they conflict with one another to compete for the same functional unit.

The programmer's view of the *logical processor* in Figure 1(b) is almost as same as the view of physical processor in Figure 1(a), although the physical organization is different. In the field of numerical computation, parallel loop execution techniques developed for multiprocessor, such as *doall* or *doacross* techniques which assign iterations to processors, are also applicable to our processor.

In this paper, we will concentrate most of our architectural interest upon parallel multithreading and restrict ourselves to simply outlining concurrent multithreading. The rest of this paper is organized as follows. In section 2, our multithreaded processor architecture is addressed. Section 3 demonstrates the effectiveness of our architecture via simulation results. In section 4, preceding works on multithreaded architectures are presented to be compared with our architecture. Concluding remarks are made in section 5.



(a) Conventional organization



(b) Organization using united elementary processors

Figure 1: Multiprocessor system organization

2 Processor Architecture

2.1 Machine Model

2.1.1 Hardware Organization

As shown in Figure 2, the processor is provided with several instruction queue unit and decode unit pairs, called *thread slots*. Each thread slot, associated with a program counter, makes up a *logical processor*, while an instruction fetch unit and all functional units are physically shared among logical processors.

An instruction queue unit has a buffer which saves some instructions succeeding the instruction indicated by the program counter. The buffer size needs to be at least $B = S \times C$ words, where S is the number of thread slots, and C is the number of cycles required to access the instruction cache.

An instruction fetch unit fetches at most B instructions for one thread every C cycles from the instruction cache, and attempts to fill the buffer in the instruction queue unit. This fetching operation is done in an interleaved fashion for multiple threads. So, on the average, the buffer in one instruction queue unit is filled once in B cycles. When one of the threads encounters a branch instruction, however, that thread can preempt the fetching operation. The instruction cache and fetch unit might become bottleneck for a processor with many thread slots. In such a case, another cache and fetch unit would be needed.

A decode unit gets an instruction from an instruction queue unit and decodes it. The instruction set is based on a RISC type, and a load/store architecture is assumed. Branch instructions are executed within the decode unit. The data dependencies of an instruction are checked using the scoreboarding technique, and issued if it is free of dependencies. Otherwise, issuing is interlocked.

Issued instructions are dynamically scheduled by *instruction schedule units* and delivered to functional units. Unless an instruction conflicts with other instructions issued from other decode units over the same

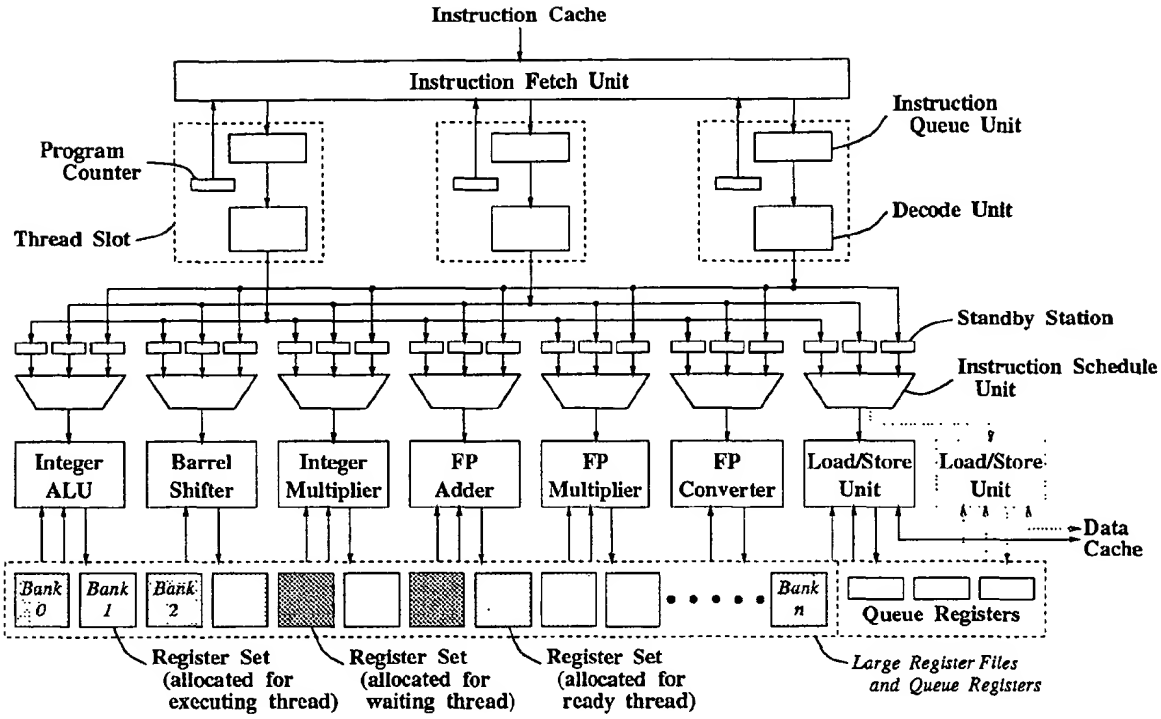


Figure 2: Hardware organization with three thread slots

functional unit, the instruction is sent immediately to the functional unit and executed. Otherwise, the instruction is arbitrated by the instruction schedule unit.

When an instruction is not selected by an instruction schedule unit, it is stored in a *standby station* and remains there until it is selected. Standby stations can allow decode units to proceed with their operations even if previously issued instructions cause resource conflicts. For example, while a shift instruction stays in a standby station, a succeeding add instruction from the same thread could be sent to the ALU. Consequently, instructions from a thread are executed out of order, though they are issued from a decode unit in order. A standby station is a simple latch whose depth is one, and differs from a reservation station in Tomasulo's algorithm[2] because issued instructions are guaranteed to be free of dependencies in our architecture.

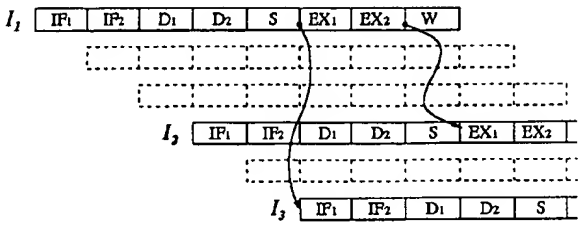
A large general-purpose register file, as well as floating-point one, is divided into banks, each of which is used as a full register set private to a thread. Each bank has two read ports and one write port. More read ports are unnecessary because operand data can be stored in standby stations. In order to support concurrent multithreading, the physical processor is provided with more register banks than thread slots. When a thread is executed, the bank allocated for the thread is logically bound to the logical processor. The exclusive one-to-one binding between a logical processor and a register bank guarantees that the logical processor does not access any register banks other than the bank bound to it. In contrast, *queue registers* are special registers which enable communications between logical processors at the register-transfer level.

Some disadvantages with our machine model include the hardware cost of register files, multiple thread slots, and dynamic instruction scheduling facilities. The increased complexity in the network between functional units and register banks is also a disadvantage.

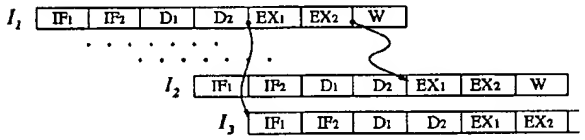
2.1.2 Instruction Pipeline

Figure 3(a) shows the instruction pipeline of the logical processor, which is a modification of a superpipelined RISC processor as shown in Figure 3(b). In the logical processor pipeline, two instruction fetch(IF) stages are followed by two decode(D) stages, followed by a schedule(S) stage, followed by multiple execution(EX) stages, followed by a write(W) stage.

The IF stages are shown for convenience, and, in fact, an instruction is read from a buffer of an instruction queue unit in one cycle. In stage D_1, the format or type of the instruction is tested. In stage D_2, the instruction is checked if it is issuable or not. Stage S is inserted for the dynamic scheduling in instruction schedule units.



(a) Instruction pipeline of multithreaded processor



(b) Instruction pipeline of base RISC processor

Figure 3: Instruction pipeline

The number of EX stages is dependent on the kind of instruction. For example, in our simulation (section 3), we assumed each operation had latencies as listed in Table 1. The result latency means the number of cycles before the instruction finishes execution (i.e. the number of EX stages), and the issue latency is the number of stages before another instruction of the same type may be issued. Since two cycles are assumed to be required to access the data cache as well as the instruction cache, the issue latency of load/store instructions is two cycles. Other instructions can be accepted in every cycle. So, the issue latency of these instructions is one cycle. In stage W, the result value is written back to a register.

Required operands are read from registers in stage S. A scoreboarding bit corresponding to a destination register is flaged on in stage S and off in the last of the EX stages. This can avoid additional delay on the initiation of data-dependent instructions. That is, assuming instruction I_2 uses the result of instruction I_1 as a source, at least three cycles are required between I_1 and I_2 in Figure 3(a). The same cycles are also required in the base RISC pipeline in Figure 3(b).

In the case of branch instructions, an instruction fetch request is sent to the instruction fetch unit at the end of stage D_1, even if a conditional branch outcome is still unknown at that point. Assume I_1 is a branch instruction and I_3 is an instruction executed immediately after I_1. In Figure 3(b), the delay between I_1 and I_3 is four cycles. But the delay in Figure 3(a) is five. What is worse, it could become more than five if some threads encounter branches at the same time. It

Table 1: Functional unit and issue/result latencies of instructions

functional unit	category of instructions	latency (cycle)	
		issue	result
Integer ALU	add/subtract	1	2
	logical	1	2
	compare	1	2
Barrel Shifter	shift	1	2
Integer Multiplier	multiply	1	6
	divide	1	6
FP Adder	add/subtract	1	4
	compare	1	4
	absolute/negate	1	2
	move	1	2
FP Multiplier	multiply	1	8
	divide	1	24
FP Converter	convert	1	4
Load/Store	load	2	4
	store	2	(4)

is obvious that the single thread performance is damaged. In our architecture, however, while some threads are blocked because of branches, other types of instructions from other threads are executed. Consequently, the parallel multithreading scheme has a potential to hide the delay of branches as well as other arithmetical operation delays, and enhance machine throughput.

2.1.3 Support of Concurrent Multithreading

Each pair of general purpose and floating-point register sets, together with an *instruction address save register* and a *thread status register*, is conceptually grouped into a single entity called a *context frame*, and allocated to a thread. An instruction address save register and a thread status register are used as save areas for the program counter and program status words (which hold various pieces of thread-specific state), respectively.

Since the processor has more context frames than thread slots, context switching is achieved rapidly by changing the logical binding between a logical processor and a context frame. As long as the number of threads to be scheduled does not exceed the number of physical context frames, threads can be resident on the physical processor, reducing the overhead to save or restore contexts to/from memory.

Another important element of the context frame is the *access requirement buffer*, which contains outstanding memory access requirements. When a thread in running state issues load/store instructions, these instructions are copied and recorded in the access requirement buffer, and deleted when they are performed.

The processor is pipelined and standby stations enable out-of-order execution of instructions. This makes

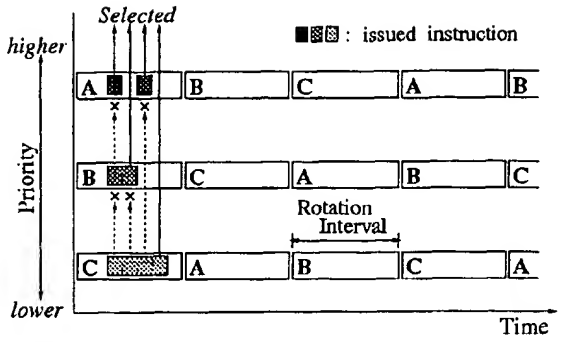


Figure 4: Dynamic instruction selection policy in instruction schedule unit (A, B, and C denote thread slots.)

the restart mechanism of programs somewhat complex. Before the thread is switched out, the logical processor should wait until all issued instructions but load/store instructions are performed. A load/store instruction is not always performed in short latency, and this is the main source of context switching in our architecture. Therefore, the load/store instructions in the execution pipeline could be flushed. But the restarting of threads is possible because such outstanding memory access requests are also saved as a piece of the context. When the thread is resumed, instructions in the access requirement buffer are decoded again and re-executed, being followed by the instruction indicated by the instruction address save register. This mechanism also ensures imprecise but restartable interrupts on page fault. Mechanisms similar to this can be applied to floating-point exception handling.

Context switching is triggered by a data absence trap. Detailed conditions concerning such traps must be considered with care, because incorrect choices can create starvation and thrashing problems. We will report our solution including cache/memory architecture in another paper in the future.

2.2 Instruction Scheduling Strategy

In this section, we present the dynamic instruction selection policy in the instruction schedule unit. Simple selection policy is preferable, so as not to enlarge the instruction pipeline pitch.

Figure 4 illustrates an example of the policy with multi-level priorities. A unique priority is assigned to each thread slot. Unless an issued instruction conflicts with other instructions from other thread slots to which higher priorities are assigned, it is selected by the instruction schedule unit. In order to avoid starvation, the priorities are rotated. The lowest priority is assigned to the thread slot which had the highest priority before the rotation.

The instruction schedule unit works in one of two modes: *implicit-rotation mode* and *explicit-rotation mode*. The mode is switched to the alternative mode through a privileged instruction. In the implicit-rotation mode, priority rotation occurs at a given number of cycles(*rotation interval*), as shown in Figure 4. On the other hand, in the explicit-rotation mode, the rotation of priority is controlled by software. The rotation is done when a *change-priority instruction* is executed on the logical processor with the highest priority.

There are two reasons why our architecture supports the explicit-rotation mode. One is to aid the compiler to schedule the code of threads executed in parallel when it is possible. The other is to parallelize loops which are difficult to parallelize using other architectures. These features are discussed in section 2.3.2 and 2.3.3, respectively.

2.3 Parallel Execution of a Single Loop

2.3.1 Overview

Parallel execution of a single loop is available on our multithreaded machine, by assigning iterations to logical processors. For example, in the case of a physical processor with n thread slots, the kth logical processor executes the $ni+k$th$(i = 0, 1, \cdots)$ iteration respectively.

The explicit-rotation mode presented in section 2.2 is one of the facilities supporting parallel execution of a loop. In this mode, a context switch due to data absence is suppressed and a physical processor is occupied entirely by the threads from a loop. In the usual case, a change-priority instruction is inserted at the end of each iteration by the compiler.

A *fast-fork instruction* generates the same number of threads as logical processors by setting its own next instruction address to program counters of other thread slots and activating them. It also sets unique logical processor identifiers to special registers corresponding to each logical processor. Each thread, therefore, can be informed which iterations it executes.

In the case of *doall* type of loops, it is unnecessary for logical processors to communicate with one another. But the execution of *doacross* type of loops requires communication between logical processors. One solution would be communication through memory. But in order to reduce the communication overhead, we provide the processor with queue registers. They are queue-structured registers connected to functional units.

The connection topology among logical processors through queue registers is important when considering the trade-offs between hardware cost and its effect. One simple and effective connection topology is a ring net-

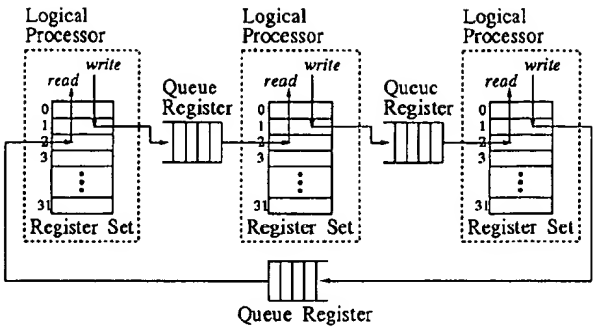


Figure 5: Logical view of queue registers

work, as shown in Figure 5. Many *doacross* type of loops encountered in practice have one iteration difference. When a loop has more than one iteration difference, data should be relayed through several logical processors. This could create overhead. Some techniques, however, are available to convert loops so that the iteration difference is reduced to one. Loop unrolling is one such technique[3].

Queue registers are enabled and disabled by special instructions. When enabled, two queue registers, for reading from the previous and writing to the next logical processor respectively are mapped into two general-purpose or floating-point registers. The reference to such registers is interpreted as the reference to queue registers, reducing data move overheads between queue registers and general/floating-point registers. *Full/empty bits* attached to queue registers are available as scoreboarding bits.

2.3.2 Static Code Scheduling

As mentioned in section 1, in the case of computer graphics programs, it is often difficult to predict the control sequence of a thread before execution, because the sequence is determined dynamically by data-dependent branches. In such cases, in order to obtain high throughput, the compiler could employ no other techniques than a simple list scheduling algorithm[4]. The compiler reorders the code without consideration of other threads, and concentrates on shortening the processing time for each thread. Short processing time of a single thread means a high issuing rate of instructions because the number of instructions is constant whether scheduled or not. Though this scheduling approach does not have control over resource conflicts, it aims at flooding functional units with sufficient instructions by parallel execution of such scheduled code.

On the other hand, in the case of numerical computation programs, it is often possible to predict the execution-time behavior of a loop. Therefore, the compiler employs more judicious code scheduling techniques.

For example, software pipelining[5, 6], which is an effective code scheduling technique for VLIW architectures, employs a resource reservation table to avoid resource conflicts. Such a technique is also applicable to our architecture. But it might postpone the issue of instructions which would cause a resource conflict. So, straight use of the algorithm could miss opportunities to issue instructions. Consequently, we have developed a new algorithm which makes the most of the function of standby stations and instruction schedule units.

Our algorithm employs not only a resource reservation table but also a standby table whose entry corresponds to a standby station. Our algorithm differs from software pipelining in the point described below. When all of the instructions without dependencies at an issuing cycle have resource conflicts, a software pipeliner would generate a NOP code. Our code scheduler, however, checks entries of the standby table for those instructions. If an entry is not marked, the instruction is determined to be issued and the entry is marked. This means the instruction is stored in a standby station. The explicit-rotation mode enables the compiler to know which instruction is selected. The resource reservation table is not used only to determine if there is a resource conflict, but also to tell the compiler when the instruction in the standby station is executed.

2.3.3 Eager Execution of Sequential Loop Iterations

Using an example, this section shows that our architecture can parallelize loops which are difficult to parallelize using vector or VLIW architectures.

The source code fragment of a sample program is shown in Figure 6. This program traverses a linear linked list. Although it is a simple example, it demonstrates the application of our execution scheme using a loop structure that is fundamental to many application programs.

```
ptr = header;
while ( ptr != NULL ) {
      tmp = a * ((ptr->point)->x)
            + b * ((ptr->point)->y) + c;
      if ( tmp < 0 )
            break;
      ptr = ptr -> next;
}
```

Figure 6: A sample program (written in C)

Each iteration is executed on a logical processor, and the value of `ptr` is passed through queue registers. The

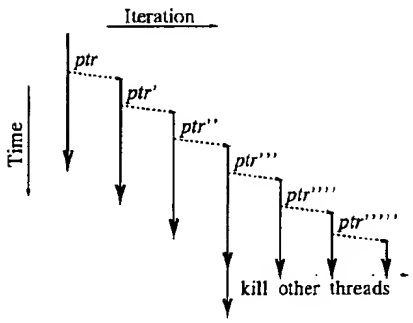


Figure 7: Parallel execution of while loop

compiler generates multiple versions for a variable ptr so that iterations can be initiated without having to wait for preceding iterations to complete. Such optimization with respect to a variable is a variation of the use of variables in dataflow machines. Figure 7 illustrates the parallel execution scheme. Each thread receives the value of ptr from the thread executed in the immediately preceding iteration, and passes the next value to the thread executed in the immediately succeeding iteration. After that, the thread continues to execute the rest of the loop body.

A thread can initiate an iteration which might not have been executed in the sequential execution. So, we call this scheme *eager execution*.

The instruction selection policy in instruction schedule units plays an important part in preserving the semantics of the sequential program. The execution of an iteration is not acknowledged until the thread gets the highest priority. It is the thread with the highest priority who executes the earliest iteration remaining at that point. By using multilevel priority, the instruction selection policy guarantees the execution of succeeding iterations does not hinder the execution of preceding iterations.

When a thread has exited from the loop, it tries to stop the operation of other thread slots and to kill other running threads. Such a special instruction is valid only for the thread with the highest priority. When a thread without the highest priority tries to execute this instruction, it is put into an interlocked state until it is given the highest priority.

If the variable tmp is global and alive after the loop, the compiler should use the special store instruction which is performed only by the thread with the highest priority. This guarantees that writes to the memory location of tmp are performed in order with respect to the source code. Some instructions are also provided to ensure consistency between contexts of threads before and after the execution of the loop.

3 Estimation

3.1 Simulation Model and Assumptions

In the following four sections, we present simulation results of our parallel multithreading architecture. Since cache simulation has not yet been implemented in our simulator, we assumed that attempts to access caches were all hit.

We used two kinds of functional unit configurations (see Figure 2). One consists of seven heterogeneous units. The other consists of those units with another load/store unit. In practice, a data cache might consist of several banks in order to be accessed simultaneously by two load/store units. But our simulation assumed there was no bank conflict. Latencies of each instruction are listed in Table 1. All machines simulated in this paper are not equipped with delayed branch mechanisms, branch prediction facilities, or overlapped register windows.

In order to estimate our architecture, we use the speed-up ratio as a criterion. It is defined as the proportion of total cycles required by multithreaded execution to those by sequential execution. Sequential execution means the execution of the sequential version of program on a RISC based processor whose instruction pipeline is shown in Figure 3(b).

3.2 Speed-up by Parallel Multithreading

This section presents simulation results of our multithreaded machine. As an application program, we have chosen a ray-tracing program, which can be easily parallelized at every pixel. The program written in C was compiled by a commercial RISC compiler with an optimization option. The object code was executed on a workstation, and traced instruction sequences were translated to be used for our simulator.

Table 2 lists speed-up ratios when the rotation interval in the instruction schedule unit is eight cycles. In the case of a processor with one load/store unit, an 1.83 times speed-up is gained by parallel execution of two instruction streams, although all of the hardware of a single-threaded processor is not duplicated in the processor. A performance improvement of a factor of 2.89 is still possible with four thread slots, though it is less effective (2.89/1.83=1.58) than the improvement attained by increasing from one to two thread slots (1.83). When eight thread slots are provided, the utilization of the busiest functional unit, the load/store unit, becomes 99%. This is the reason why the speed-up ratio is saturated at only 3.22 times. Addition of another load/store unit improves speed-up ratios by 10.4~79.8%.

Table 2: Speed-up ratio by parallel multitthreading

no. of thread slots	with one load/store unit		with two load/store units	
	without standby station	with standby station	without standby station	with standby station
2	1.79	1.83	2.01	2.02
4	2.84	2.89	3.68	3.72
8	3.22	3.22	5.68	5.79

Standby stations improve the speed-up ratio by 0~2.2%. This low improvement is due to poor parallelism within an instruction stream. In the case of application programs whose thread is rich in fine-grained parallelism, greater improvement can be achieved.

We also simulated a machine whose thread slots are provided with private instruction caches and instruction fetch units. Such organization does provide a slight speed-up. Achieved speed-up ratios are the same as those in Table 2, except that 1.79 and 5.79 is respectively replaced by 1.80 and 5.80. This shows that the delay due to instruction fetch conflicts is hidden and sharing an instruction cache between thread slots is possible.

We also examined the execution cycles with various rotation intervals (2^n cycles, where n is 0~8). But, in this application program, rotation interval did not have much influence on the performance, though using eight or sixteen cycles seems slightly superior.

3.3 Multithreading with Superscalar Design

In our simulator, each thread slot can support multiple instruction issuing from a single instruction stream similar to superscalar architectures. In this section, using the same program as in section 3.2, we discuss hybrid architectures which employ both coarse- and fine-grained parallelism.

A hybrid processor is represented here by (D,S)-processor, where D is the maximum number of instructions which a thread slot issues each cycle, and S is the number of thread slots. For a fair comparison, the hardware cost of (d,s)-processor is almost as same as that of (1,$d\times s$)-processor. For example, the instruction fetch bandwidth is $d\times s$ words per cycle in both processors. Although ($d\times 2$,$s/2$)-processor is provided with a half register set of (d,s)-processor, it requires double the number of register ports. Each thread slot checks the dependencies of instructions in the instruction window of size D, and issues instructions without dependencies every cycle. The instruction window is filled every cycle. Some techniques[7, 8] are proposed to boost the superscalar performance, but they are not provided for the thread slot because they requires additional hardware.

Table 3: Tradeoff between speed-up and employed parallelism (speed-up ratio)

total no. of issued instructions ($D\times S$)	no. of issued instructions from each thread slot (D)			
	1	2	4	8
2	2.02	1.31	—	—
4	3.72	2.43	1.52	—
8	5.79	4.37	2.79	1.66

Table 3 lists the simulation results. Values in the table are speed-up ratios. Simulated processors are composed of eight functional units. We used the instruction pipeline shown in Figure 3(b) for a (d,1)-processor, and the pipeline shown in Figure 3(a) for others. This table demonstrates that the increase of S produces a more significant speed-up than the increase of D.

The main objective of multithreading is to enhance the machine throughput. On the other hand, the main objective of superscalar architecture is to improve the single thread performance. This difference of objectives makes a simple comparison of the two architectures impossible. But, from the viewpoint of cost-effectiveness, it is clear that D=1 is the best choice if parallel multithreading is introduced into a processor architecture.

3.4 Effect of Static Code Scheduling

We will compare static code scheduling strategies discussed in section 2.3.2. The sample program is the Livermore Kernel 1. The source code is shown in Figure 8.

```
      DO 1 K = 1,N
1     X(K) = Q+Y(K)*(R*Z(K+10)+T*Z(K+11))
```

Figure 8: Livermore Kernel 1 (written in FORTRAN)

Table 4 lists average execution cycles for one iteration. The source code was compiled by a commercial compiler, and the generated object code was scheduled by our code schedulers. The simulated machine has one load/store unit. Strategy A represents a simple list scheduling approach, and strategy B represents the list scheduling approach with a resource reservation table and a standby table.

Strategy B is overall superior to other strategies. It achieves the performance improvement by 0~19.3%

Table 4: Comparison of static code scheduling

no. of thread slots	optimization strategy		
	non-optimized	strategy A	strategy B
1	50	42	—
2	25	21.5	21
3	16.7	15	14
4	12.75	11.25	11
5	10.4	9	9
6	8.83	8	8
7	8.125	8	8
8 ~	8	8	8

(unit: cycle)

over non-optimized code. The differences between effectiveness of strategy A and strategy B, however, are small in this program. Although strategy A achieves nearly the same effectiveness at a lower cost, strategy B should be employed in order to optimize the program.

The object code contains three load instructions and one store instruction, so at least (3+1)×2=8 cycles are required for one iteration. This is the reason of performance saturation.

3.5 Effect of Eager Execution

In section 2.3.3, the eager execution scheme of sequential loop iterations is addressed. This section reports simulation results of the execution of the sample program shown in Figure 6. The simulated machine has one load/store unit.

The execution of the sequential version of the object code requires 56 cycles for one iteration. Table 5 lists the average execution cycles of the parallel version for one iteration. With a small number of thread slots, when a new value of `ptr` is passed to the next thread through a queue register, the thread is still executing a preceding iteration. An increase in the number of thread slots enables a thread to initiate an iteration as soon as the data is available, but the speed-up ratio is limited by the inter-loop dependency of `ptr`.

Since parallel code includes more overhead in loop headers and footers than sequential code, sequential execution performs better than parallel execution when the number of iterations is small. As the number of iterations increases, the maximum speed-up ratio becomes closer to 56/17=3.29 times.

4 Related Works

The concept of concurrent multithreading is discussed in HEP[9], MASA[10], Horizon[11], and P-RISC[12].

Table 5: Evaluation of eager execution of sequential loop iterations

no. of thread slots	execution cycle for one iteration
2	32.5
3	21.67
4 ~	17

These processors support cycle-by-cycle interleaving of instructions from multiple instruction streams. In order to improve poor performance of single thread execution in these machines, APRIL[13], and [14] support block interleaving in which a thread keeps control until it encounters a remote memory access. The hybrid data-flow/von Neumann machine[15] also employs the latter type of interleaving. Our concurrent multithreading scheme builds on such a block interleaving architecture. In our architecture, however, combination with parallel multithreading enables other threads to continue working during context switches.

Farrens and Pleszkun[16] take a multithreading approach to increase instruction issuing rate. This idea is closely allied to the basic one of our parallel multithreading. In their architecture, two instructions from two threads compete with each other to be issued, but the total number of issued instructions each cycle is still at most one. On the other hand, in our architecture, multiple instructions can be issued unless there are resource conflicts.

Daddis and Torng[17] propose a superscalar architecture which uses multiple threads to increase functional unit utilization. Prasadh and Wu[18] present a VLIW architecture which does the same. In our opinion, such hybrid architectures are not as cost-effective as a multithreaded non-superscalar architecture as mentioned in section 3.3. Functional unit utilization is an important factor, but the instruction fetch bandwidth sufficient to sustain multiple functional units must also be considered. For example, although Prasadh's four-threaded, eight-functional-unit processor achieves about eight times speed-up, it requires an instruction fetch bandwidth of thirty-two words per cycle.

5 Concluding Comments

In this paper, we proposed a multithreaded architecture oriented for use as a base processor of multiprocessor systems. Through the simulation using a ray-tracing program, we ascertained that a two-threaded, two-load/store-unit processor achieves a factor of 2.02 speed-up over a sequential machine, and

a four-threaded processor achieves a factor of 3.72 speed-up. We also examined the performance of multithreaded architecture combined with superscalar technique. This evaluation, however, shows the best cost-effectiveness is achieved when a processor employs only coarse-grained parallelism and ignores fine-grained parallelism.

We developed a static code scheduling algorithm applicable for our architecture, which is derived from idea of software pipelining. We also investigate possibilities for broadened use of parallel multithreading scheme. One instance of such intention is presented as an eager loop execution scheme. It parallelizes loops which are difficult to be parallelized with other architectures.

One weak point of this paper is the poor variety of tested programs. We should confirm the effectiveness of our architecture by using many other application programs. We are currently working on evaluating finite cache effects and the detailed design of the processor.

References

[1] H. Hirata, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishizawa, "A Multithreaded Processor Architecture with Simultaneous Instruction Issuing," In *Proc. of Intl. Symp. on Supercomputing, Fukuoka, Japan*, pp. 87-96, November 1991.

[2] R.M.Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," In *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25-33, January 1967.

[3] A. Munshi and B. Simons, "Scheduling Sequential Loops on Parallel Processors," In *IBM Technical Report 5546*, 1987.

[4] E.G. Coffman, In *Computer and Job-Shop Scheduling Theory*, 1976.

[5] R.F. Touzeau, "A Fortran Compiler for the FPS-164 Scientific Computer," In *Proc. of the ACM SIGPLAN'84 Symp. on Compiler Construction*, pp. 48-57, June 1984.

[6] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," In *Proc. of the SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pp. 318-328, June 1988.

[7] K. Murakami, N. Irie, M. Kuga and S. Tomita, "SIMP (Single Instruction Stream / Multiple Instruction Pipelining): A Novel High-Speed Single Processor Architecture," In *Proc. of the 16th Annual Intl. Symp. on Computer Architecture*, pp. 78-85, May 1989.

[8] M.D. Smith, M.S. Lam and M.A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pp. 344-354, May 1990.

[9] B.J. Smith, "A Pipelined, Shared Resource MIMD Computer," In *Proc. of the 1978 Intl. Conf. on Parallel Processing*, pp. 6-8, August 1978.

[10] R.H. Halstead and T. Fujita, "MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing," In *Proc. of the 15th Annual Intl. Symp. on Computer Architecture*, pp. 443-451, June 1988.

[11] M.R. Thistle and B.J. Smith, "A Processor Architecture for Horizon," In *Proc. of 1988 Supercomputing Conf.*, pp. 35-41, November 1988.

[12] R.S. Nikhil and Arvind, "Can Dataflow Subsume von Neumann Computing?" In *Proc. of the 16th Annual Intl. Symp. on Computer Architecture*, pp. 262-272, June 1989.

[13] A. Agarwal, B.H. Lim, D. Kranz and J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing," In *Proc. of the 17th Annual Intl. Symp. on Computer Architecture*, pp. 104-114, June 1990.

[14] W.D. Weber and A. Gupta, "Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results," In *Proc. of the 16th Annual Intl. Symp. on Computer Architecture*, pp. 273-280, June 1989.

[15] R.A. Iannucci, "Toward a Dataflow/von Neumann Hybrid Architecture," In *Proc. of the 15th Annual Intl. Symp. on Computer Architecture*, pp. 131-140, June 1988.

[16] M.K. Farrens and A.R. Pleszkun, "Strategies for Achieving Improved Processor Throughput," In *Proc. of the 18th Annual Intl. Symp. on Computer Architecture*, pp. 362-369, May 1991.

[17] G.E. Daddis Jr. and H.C. Torng, "The Concurrent Execution of Multiple Instruction Streams on Superscalar Processors," In *Proc. of the 20th Intl. Conf. on Parallel Processing*, pp. I:76-83, August 1991.

[18] R.G. Prasadh and C. Wu, "A Benchmark Evaluation of a Multi-Threaded RISC Processor Architecture," In *Proc. of the 20th Intl. Conf. on Parallel Processing*, pp. I:84-91, August 1991.